12

TECHNICAL REPORT O-79-1

IMPLEMENTATION AND EVALUATION OF INTERVAL ARITHMETIC SOFTWARE

Report 4

THE IBM 370, DEC 10, AND DEC PDP-11/70 SYSTEMS

by

Ronnie G. Ward

Department of Computer Science
University of Texas at Arlington
Arlington, Tex. 76019

April 1979

Report 4 of a Series

Approved For Public Release; Distribution Unlimited

Prepared for Office, Chief of Engineers, U. S. Army
Washington, D. C. 20314
Under Contract Nos. DACA39-76-M-0247 and DACA39-76-M-0356
Monitored by Automatic Data Processing Center
U. S. Army Engineer Waterways Experiment Station
P. O. Box 631, Vicksburg, Miss. 39180

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report 0-79-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER TR-O-79-1
4. TITLE (and Subtitle) IMPLEMENTATION AND EVALUATION OF INTERVAL ARITHMETIC SOFTWARE. Report 4: The IBM 370, DEC 10, and DEC PDP-11/70 Systems.		5. TYPE OF REPORT & PERIOD COVERED Report 4 of a series
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ronnie G. Ward		8. CONTRACT OR GRANT NUMBER(s) Contract Nos. DACA39-76-M-0247, DACA39-76-M-0356
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Texas at Arlington. Dep~~artment~~t of Computer Science ~~Arlington, Tex. 76019~~		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Integrated Software Research & Development Program, AT11
11. CONTROLLING OFFICE NAME AND ADDRESS Office, Chief of Engineers, U. S. Army Washington, D. C. 20314		12. REPORT DATE Apr 1979 (12) 75p.
		13. NUMBER OF PAGES 70
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) U. S. Army Engineer Waterways Experiment Station Automatic Data Processing Center Vicksburg, Miss. 39180		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Arithmetic Computer systems programs Evaluation Interval arithmetic		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This is Report 4 of a series entitled "Implementation and Evaluation of Interval Arithmetic Software." The series concerned implementation and evaluation of an interval arithmetic software package on six different computer systems. The other reports to be published in the series are: Next page → (Continued)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

411247

20. ABSTRACT (Continued).

Report 1: The State of the Interval: Evaluation and Recommendations
Report 2: The Honeywell MULTICS System
Report 3: The Honeywell G635 System
Report 5: The CDC CYBER 70 System

Using interval arithmetic as prescribed in the Interval II package is simple because of the AUGMENT preprocessor. Computer runs illustrate that interval arithmetic can show the instability of an algorithm for a given set of data. On the other hand, interval arithmetic can establish a high level of confidence in an algorithm for a given set of data.

Interval arithmetic should be used (as any tool would be) where accuracy is of critical importance.

Interval arithmetic is expensive to use in terms of computer time, main storage, and personnel time spent in error analysis.

Some reasonable means of estimating the cost of using interval arithmetic in a given situation should be developed. These costs would be important in the decision process of determining whether or not interval arithmetic would be worth the effort or not.

Techniques of accuracy extension on short word length machines should be examined.

PREFACE

In December 1975, the Automatic Data Processing (ADP) Center of the U. S. Army Engineer Waterways Experiment Station (WES), Vicksburg, Miss., submitted a proposal to implement and evaluate interval arithmetic, a software system for digital computer numerical analysis, on the Corps of Engineers' primary engineering computer--the WES Honeywell G635. The proposal was later expanded to include the implementation and evaluation of an interval arithmetic software package on six different computer systems. Engineering and scientific data problems were selected to be used on each of the six computers with the interval arithmetic software.

The work was funded by the Office, Chief of Engineers, U. S. Army, through the Integrated Software Research and Development (ISRAD) Program, AT11, Engineering Software Research.

This is Report 4 of a series entitled "Implementation and Evaluation of Interval Arithmetic Software." The other reports to be published in the series are:

Report 1:	The State of the Interval: Evaluation and Recommendations
Report 2:	The Honeywell MULTICS System
Report 3:	The Honeywell G635 System
Report 5:	The CDC CYBER 70 System

This report was written by Dr. Ronnie G. Ward of the Department of Computer Science, University of Texas at Arlington. His work was performed under Contract No. DACA39-76-M-0247, dated 28 April 1976, and Contract No. DACA39-76-M-0356, dated September 1976, and through support from the University of Texas at Arlington supplied directly through organized research funds. (The project also benefitted from the work of Dr. Darrell Ward at the University of Texas Health Science Center.) The work concerned implementation and evaluation of an interval arithmetic software system on the IBM 370, DEC 10, and DEC PDP-11/70 computer systems.

Dr. J. Michael Yohe, Director of Academic Computing Services, University of Wisconsin-Eau Claire, developed and wrote the interval arithmetic software package which was implemented on each of the six computer systems. Dr. Fred D. Crary, formerly with the U. S. Army Mathematics Research Center, University of Wisconsin-Madison, developed and wrote the AUGMENT precompiler which was implemented on each computer system as a front-end to the interval arithmetic software package. Dr. Crary also prepared a series of remarks pertinent to items noted by Dr. Ward in this report. These remarks are presented in Appendix A; the work in preparing them was performed under Contract No. DAAG29-75-C-0024. Dr. Yohe and Dr. Crary are specially thanked and recognized for their technical contributions and assistance.

Mr. James B. Cheek, Jr., formerly with the ADP Center, WES, provided initial impetus and guidance for the project. Mr. Fred T. Tracy, ADP Center, WES, provided expert advice and technical guidance during the project. Dr. N. Radhakrishnan, Special Technical Assistant, ADP Center, furnished technical guidance and general project supervision. The project and the report were monitored by Mr. William L. Boyt under the general supervision of Mr. D. L. Neumann, Chief of the ADP Center.

Directors of WES during the project and the preparation of the report were COL G. H. Hilt, CE, and COL J. L. Cannon, CE. Technical Director was Mr. F. R. Brown.

Copies of the other reports of the series, computer listings of the interval program and of AUGMENT for each computer system, and runs of the benchmarks for each computer system may be obtained from the ADP Center, WES.

CONTENTS

PART I: IMPLEMENTATION OF THE 'AUGMENT' PRECOMPILER AND INTERVAL ARITHMETIC ON THE IBM 370, DEC 10, AND DEC PDP-11/70 SYSTEMS

1. Introduction

Organization of this Part

Phase I of this project was concerned with installing the AUGMENT preprocessor [1,2] and the interval arithmetic package [5,7] on three computer systems -- the DEC System 10, DEC PDP-11/70, and an IBM System 370. This Part describes results of this Phase I activity. Section 2 discusses the experiences and decisions in implementing AUGMENT on the target machines. Section 3, in a similar manner, presents the approach and difficulties in implementing the interval arithmetic package on the three systems. Section 4 discusses the use of the packages from a programmer's viewpoint. Limitations of AUGMENT and possible pitfalls in using the interval package are discussed. Section 5 discusses some recommendations and conclusions.

Summary of Phase I Activities

Accomplishments -

* AUGMENT was successfully implemented on the DEC System 10 and the IBM System 370.
* The Interval Arithmetic package was implemented on all three target computer systems.
* AUGMENT and the Interval Package have been tested for correct operation.
* Experience has been gained in using the packages. Possible pitfalls are discussed.

Problems -

* Due to storage requirements, AUGMENT could not be implemented on the DEC PDP-11/70.
* The logistics of working with three separate computer systems simultaneously proved to be a serious obstacle.
* Debugging the interval arithmetic primitives in a batch environment is much more difficult than debugging them on a timesharing system.

2. AUGMENT Implementation

General Information

The implementation of AUGMENT on a computer system is straightforward. Eight machine dependent routines [3] must be coded and executed through a test driver supplied with AUGMENT. The major problem encountered in implementing the system was that of working on three separate computer systems simultaneously. The systems are not physically located at the same site, and they are not compatible in any respect. This compounds the problem of transferring material betw-en the systems since it has to be hand carried and specially processed on each system.

To illustrate this point, the PDP-11/70 has no software utility to read foreign tapes. So the choices were to write such a utility or punch AUGMENT into cards for loading on the 11. A utility was written due to the volume of cards involved.

Implementation on the IBM System 370

The distribution tape containing AUGMENT, the text for the primitives, and a sample program for AUGMENT processing was received. The last blocks on all three files were "short" blocks, and contained '@@@@' as the last four characters. This caused some problems in procesing the tape. Since AUGMENT has been installed on a System 370 prior to this project, primitives already existed and merely had to be checked out.

Subroutine PACK contained a syntax error (unbalanced parens); MOVHOL incorrectly referenced MINO instead of MIN0; PACK incorrectly referenced 'CHARS' instead of 'CHAR'; ORDER documentation noted the problem with an overflow when testing the relative orders which could have been circumvented very cleanly using a logical IF (compare instruction generated) rather than an arithmetic IF (subtract instruction generated). Finally, the documentation for STRWDS [3] is ambiguous (because of the "if's"). It appears that the routine can be called under different conditions. This is confusing to someone who does not understand AUGMENT internals.

The primitives, after some modification, passed the acceptance testing of the driver routine supplied. It should be noted that this routine performs only a cursory check of the primitives. However, a more extensive testing routine is probably not needed since the primitives are so basic to the preprocessor that any errors in them will likely show up rapidly with any use of AUGMENT. In the interest of completeness, it should also be noted that AUGMENT routines MAIN and MOVNUM do not end with allowable executable statements on IBM's Fortran H compiler.

AUGMENT was installed on an IBM System 370 running under OS/MVT. Since job scheduling under this operating system is based in part on core requirements it was decided to overlay AUGMENT as described in [3]. Without overlaying, AUGMENT would require 306K bytes of storage using the Fortran G1 level compiler. With overlaying the root phase requires 84K, the description phase 76k, and the process phase 146K. This comes to a total of 230K needed at any one time for AUGMENT. Hence a region size of 230K is required. Thus, AUGMENT is executed as a class C job rather than class D improving turnaround significantly. Explicit calls to effect the overlaying are not needed since the OS/MVT linkage editor [18] inserts code to manage the overlay structure at run time. It should be noted that linking AUGMENT into an overlay structure required 384K bytes of storage for the linkage editor. This meant that AUGMENT could only be linked when the computer was relatively idle.

One other point is that upgrading AUGMENT to another version, for example Version 4I to 4J, is rather simple provided the source files for each of three overlay phases are compiled and linked separately. A final link is required to create an executable AUGMENT load module. Experience has shown that changes can be made more rapidly if the system is maintained as described.

AUGMENT was installed on an IBM System 370 running under OS/MVT. Since job scheduling under this operating system is based in part on core requirements it was decided to overlay AUGMENT as described in [3]. Without overlaying, AUGMENT would require 306K bytes of storage using the Fortran G1 level compiler. With overlaying the root phase requires 84K, the description phase 76k, and the process phase 146K. This comes to a total of 230K needed at any one time for AUGMENT. Hence a region size of 230K is required. Thus, AUGMENT is executed as a class C job rather than class D improving turnaround significantly. Explicit calls to effect the overlaying are not needed since the OS/MVT linkage editor [18] inserts code to manage the overlay structure at run time. It should be noted that linking AUGMENT into an overlay structure required 384K bytes of storage for the linkage editor. This meant that AUGMENT could only be linked when the computer was relatively idle.

One other point is that upgrading AUGMENT to another version, for example Version 4I to 4J, is rather simple provided the source files for each of three overlay phases are compiled and linked separately. A final link is required to create an executable AUGMENT load module. Experience has shown that changes can be made more rapidly if the system is maintained as described.

AUGMENT's source was maintained on tape at the SYSTEM 10 due to the large number of disk blocks required to store it on-line. TRIM was used to knock off the sequence fields of the card images, and PIP with a '/T' switch removed trailing blanks. This reduced the disk storage requirements by 60%, but deleting trailing blanks invalidated Hollerith constants containing trailing blanks in DSCRIB (AUG30500) and TRNSCD (AUG38900).

The F10 compiler was used on the 10 rather than the F40 compiler due to the differences in the quality of the object code produced. This made the FORDDT debugger available which makes program check out very simple. Using F10 on AUGMENT established a pattern of using this compiler throughout the project.

AUGMENT required no overlays on the DEC-10 since a virtual operating system was being used. A total of 109 sharable and 13 non-sharable 512 word pages are needed for AUGMENT which has a low paging rate.

Since no overlaying was used, the entire AUGMENT system is compiled in one execution. Under the F40 compiler, the multiple BLOCK DATA subprograms in AUGMENT caused a failure during the linking operation. Under F10 this problem does not exist. However, F10 balked at the IF statement (AUG01530) in routine CCNVRT. The CONTINUE on the IF was changed to J=J to circumvent the problem. Note that the function call contained in the IF changes AUGMENT's state. So the F cannot be deleted. In other words, the IF accomplishes something useful even though it appears to be unnecessary. F10 also puts out warning messages about modifying a DO index in routines GETSYM (AUG74190), CLEAR (AUG03255), ENDIT (AUG22930, and AUG22945), and INDEX (AUG30790). These are warnings and can be suppressed easily.

Implementation on the DEC PDP-11/70

A first step in the implementation of the AUGMENT preprocessor on the PDP-11 system was to install and check out the eight machine-dependent routines in the AUGMENT system (ORDER, MOVHOL, CCODE, NUMIN, STRWDS, STRCHR, PACK, STRLNG). The machine dependencies on the PDP-11 were mainly in the handling of A-TYPE (character) data. The PDP-11 Fortran-F4P (extended Fortran) compiler was available for use. The handling of characters and Hollerith constants under this compiler was found to be very unique. The 16 bit words are used to store two characters and are filled from the low-order end. Thus a two character Hollerith constant would be stored with the first character in the lower half of the word, and the second in the upper. This condition required modifications in the packing and unpacking of characters in the above routines. Also the indexing of the characters in a string could not be linear as the storage locations did not coincide with the order of the characters in the string. After a study of the manuals and using test programs to check the handling of character data, the routines were implemented and tested using the test driver supplied. As with the System 10, the ENCODE/DECODE statements were used in implementing the primitives. Unlike the 10, however, only one record can be read with these statements on the 11. Therefore, an explicit loop was built around these statements making the code more lengthy.

With the machine-dependent features of the preprocessor installed, the next step was to implement the entire AUGMENT system. The source for the routines was transferred to the 11 via magnetic tape and each was compiled separately giving a listing containing the size of the object code for each. The total of the individual routines came to approximately 60K words, thus making evident that an overlay structure must be developed to allow the installation of the system on the 11, which requires a task be no larger than 32K words.

Since no cross-reference of the calling sequence for the routines was provided, it was necessary to obtain one. This was done through the DEC-10 implementation where the routines were also installed. The cross-reference, along with the source code, was used to find natural breaking points in the flow of logic for the overlay purposes.

The root segment for the structure was arrived at first, and consisted of the main routine; the COMMON blocks, which were located in the

BLOCK DATA subroutine; and the eight machine dependent routines. The BLOCK DATA subroutine consisted of a combination of the two original BLOCK DATA's and the other COMMON blocks used in various other routines not formerly in either BLOCK DATA. These other COMMON blocks were added to the BLOCK DATA subroutine to allow all of the COMMON blocks to reside in the root segment of the overlay structure. An initial break down of the processor was found at the 'INITAL', 'DESCON', and 'PROCSS' routines called directly from MAIN. it was then necessary for each of these three segments to contain all of the routines which could be called from each of the three routines. Working with the cross-reference alone proved to be too difficult when proceeding through several levels of calls seeking the routines needed for each segment. It was therefore found that a more extensive calling sequence analysis was needed.

A program was written in PL/I which would accept, as input, the current cross-reference consisting of the name of each routine, its size, and the names of the routines it called directly. The program processed the cross-reference by building a structure which was then traversed producing an exhaustive listing of the routines and all routines which could possibly be called from each. The output consisted of the above information for every routine, and the total size of all of the routines listed as possibly being called. This revealed some apparent (though fortunately not real) recursion in AUGMENT between routines QUIT and MOVNUM.

The complete analysis provided by the above program considerably reduced the task of developing an overlay structure, and after analysis of the original cross-reference and the above multi-level calling sequence, a structure was arrived at for the overlay. The structure consisted of the aformentioned breakdown below the main routine with each of the three segments further broken down. Since the amount of overhead incurred by the use of overlay on the 11 was not certain, the sizes of the segments were computed soley on the basis of the size of each routine with an estimated amount of overhead.

When the overlay description was used in an attempt to link the object modules into a executable task the overlay description was found to contain too many names. Therefore the description had to be reduced by removing common names from segments, where possible, and moving them to a point in the description such that they would be path-loaded (see IAS TASK BUILDER reference manual p.6-4). The search for the common routines was made using the listings of the routines contained in each segment and working from the outward segments toward the root, moving common names to a point above the segments containing them. Upon completion of this intersection of the segments, the overlay description was reduced by approximately 50% and another attempt was made at task building. When the overlay description was processed the task overflowed the 32K limit and the need for a further breakdown of the structure was evident.

Working from the points in the structure where overflow was taking place, the description was again revised to the greatest possible breakdown as shown in Figure 1 (refer to the IAS TASK BUILDER reference manual p.5-6 for a description of the overlay description language). Another attempt at task building again resulted in overflow.

At this point the overlay description was specifying several overlays for the processing of a single input card by the AUGMENT system, and the breakdown of the structure was at its lowest point. Therefore, an attempt was made, through compilation options, to reduce the sizes of the individual routines. The code generated by the compiler to facilitate trace back information for subprogram calls was suppressed and other options were used in an attempt to minimize the size of the routines. An attempt task build at this point still produced overflows. Further reductions in storage requirements were made at task-build time by reducing the Fortran I/O unit numbers to 1-4; the number of active files to 3; and the file control system buffer size. Task building failed again however.

Any further reduction of the task size through a further breakdown of the overlay structure would render the preprocessor infeasable because of the amount of time spent on overlaying during the execution of the system. It would therefore seem that the next step would require modifications to the AUGMENT source code to reduce the number of routines by eliminating calls to routines and placing the code for them in line. A scan of the AUGMENT source revealed the following routines as candidates: GETP, OHFNDC, PUTP, GETL, FHHEAD, OHHEAD, SFENCE, CFENCE, CHHEAD, CLRNUM, CLTEMP, COLMNS, DMCLR, DMSET, SMWIPE, RFNEXT. Each routine consist of one or two assignment statements, and/or a single call to another routine. Modifications to the source would require first analysis of the flow of logic in the AUGMENT system and due to the time element this could not be considered, therefore the installation of the AUGMENT preprocessor on the PDP-11 was abandoned. However, since AUGMENT produces standard Fortran, the System 370 and DEC-10 versions of AUGMENT can be used as host processors for the PDP-11/70.

Another consideration in AUGMENT implementation is the word size used to represent integers. The PDP-11/70 supports 32 bit integers, but this of course requires additional storage over 16 bit integers. There is possibly a problem in routines DOESN and DOISN with external and internal statement numbers since 16 bits will only represent values up to 32,768. This value may not be large enough to accommodate these numbers.

```
       .ROOT MAIN-ORDER-MOVHOL-CCODE-NUMIN-STRWDS-STRCHR-PACK-STRLNG-BDATA-R1
R1:    .FCTR QUIT-MOVNUM-(C1,D1,P1,*SFENCE)
C1:    .FCTR PUTP-PSTOR-PUTI-CLASS-TSTOR-TREINS-TREFND-ALPHA-(I1,D1)
I1:    .FCTR *INITIAL-TPCONV-ODMAKE-STENTR-FDMAKE-TPMAKE-OHMAKE-FHMAKE
D1:    .FCTR *DESCON-FDCODE-GETI-GETL-GETP-GETSYM-NTHR-NXTSTG-PRNTCD-D2
D2:    .FCTR         READCD-SMATCH-TPFIND-TPFND-TPFNDH-TREFST-TRENXT-ECHO-D3
D3:    .FCTR         ERRADD-GETTYP-NAMEOK-NXTELT-REFNAM-RFFIND-RFMAKE-D4
D4:    .FCTR         SEQUAL-RCLEAR-DSCRIB-USESTP-STENTR-TPMAKE-TPTEST-D5
D5:    .FCTR         CNFLCT-(TST1,ENV1)
TST1:  .FCTR *TSTCRD-FDFIND-FHFIND-FDMAKE-FHMAKE
ENV1:  .FCTR *ENVADD-(TRN1,OPR1,COM1)
TRN1:  .FCTR *TRNSCD-FDFLDG-ODCODE-OTYPES-OHHEAD-(*DMOPER,DMTYP1,DMF1)
DMF1:  .FCTR *DMFUNC-FDARGS-FHHEAD
OPR1:  .FCTR *OPRCRD-ODFIND-OHFNDC-OHFNDS-ODMAKE-OHMAKE-OPEROK
COM1:  .FCTR FDFIND-FHFIND-FDMAKE-FHMAKE-FCNALL-FCNGEN-(CNV1,COM2)
CNV1:  .FCTR *CNVCRD-CNVFCN-HIGHER-TPCONV
COM2:  .FCTR FCNARG-(*FCNCRD,*FLDCRD-FDFLDP)
P1:    .FCTR *PROCSS-OTCONT-EQUATE-PSTOR-CHKNUM-MKUSED-(CDI1,SPR1)
CDI1:  .FCTR *CARDIN-ALPHA-CLASS-GETI-GETL-GETP-GETSYM-PRNTCD-CDI2
CDI2:  .FCTR         PUTI-PUTP-READCD-SMATCH-TPFIND-TPFND-TPFNDH-CDI3
CDI3:  .FCTR         TREFST-TRENXT-TSTOR-CHHEAD-COPYCD-DMCOMN-DMSMTB-CDI4
CDI4:  .FCTR         DMTYP2-PROCRD-RFNEXT-SMHEAD
SPR1:  .FCTR *SUPER-(*CLTEMP-*CLRSTK,DCI1,COM3)
DCI1:  .FCTR *DCIMPL-ALPHA-NXTCHR-TPFIND-TPFND-LTRNUM
COM3:  .FCTR CLASS-GETI-GETP-TRENXT-(*SETXQT,TSTOR-(END1,COM4))
END1:  .FCTR *ENDIT-CCLEAR-TREFST-CFENCE-CLEAR-CLRNUM-DECASM-DECEND-END2
END2:  .FCTR        DECNAM-DECOUT-DECSTR-DMCOMN-DMSMTB-DMTYP2-ESN-INDEX-END3
END3:  .FCTR        OUTLNF-RFERAS-RFNEXT-SMHEAD-SMWIPE-USED-VERSN-RETGEN-END4
END4:  .FCTR        GETL-GETSYM-PUTI-CHHEAD-COLMNS-COPYCD-CSTYPE-FDNAME-END5
END5:  .FCTR        LOADA-MOVPTR-MOVTMP-ORNAME-OUTLIN-OUTPTR-OUTSTR-END6
END6:  .FCTR        RETEMP-RFLINK-STOREA-TYPEOF
COM4:  .FCTR ALPHA-NXTCHR-NXTSTG-TREFND-TREINS-ICNVRT-SMFIND-COM5
COM5:  .FCTR SMMAKE-(DCT1,COM6)
DCT1:  .FCTR *DCTYPE-GETSYM-ADDIMN-PUTI-PUTP
COM6:  .FCTR IMPLCT-LTRNUM-(*DCSUBR-*COPYCD,PUTI-(COM7,DCF1),CMN1)
DCF1:  .FCTR *DCFCN-PUTP-STENTR
COM7:  .FCTR GETSYM-(DCXTRN-PUTL-GETL,ADDIMN-PUTP-(*DCDIMN,COM8))
COM8:  .FCTR CHFIND-CHHEAD-CHMAKE-(*LCLGBL,DCCOMN-CADD)
CMN1:  .FCTR GETL-OHFNDC-OHFNDS-SMATCH-OUTLIN-OUTSTR-SCANNR-STRPTR-CMN2
CMN2:  .FCTR (*STDO-CPYREM,CMN3)
CMN3:  .FCTR CNVFCN-FDCODE-FDFIND-FHFIND-GETSYM-HIGHER-ODCODE-ODFIND-CMN4
CMN4:  .FCTR CCNVTR-COLMNS-COMPIL-COPYCD-CSTYPE-FDNAME-GENEXP-GENOPR-CMN5
CMN5:  .FCTR MOVPTR-MOVTMP-NCLOSE-NTRLIN-ORNAME-OUTPTR-PARSE-PCNVRT-CMN6
CMN6:  .FCTR PFETCH-PMATCH-POP-PUSH-RETEMP-RFLINK-STFCN-STOREA-TYPEOF-CMN7
CMN7:  .FCTR PUTI-CNVFCN-(*DCSTFN-PUTP-STENTR-DMCLR-DMMAKE-DMSET,STF1)
STF1:  .FCTR *STIF-TPFIND-TPFND-LOADA-(*IFINIT,RECOG-STC1)
STC1:  .FCTR FDFLDG-OTTYPES-CLRSUB-CLTEMP-*GENTST-OTGOTO-*CPYREM-STC2
STC2:  .FCTR (*GENCOD-PUTI-(*STCALL,*STREPL),*STIO,*STRETN)
       .END
```

Figure 1. PDP-11/70 AUGMENT Overlay Description

Testing

Testing of the AUGMENT primitives was accomplished using the test driver supplied with AUGMENT. On the System 370, the primitives were checked through hand simulation. On the DEC systems', the primitives were verified using dynamic debuggers [12,13]. Testing of AUGMENT itself was done using the test program supplied with AUGMENT. No problems were encountered in processing this program. There was, however, a misunderstanding in that the AUGMENT output from this test is not executable. The results of testing the primitives is included in the appendix.

During use of AUGMENT several problems were noted. Tables for blank common were not being initialized properly and since the System 370 does not clear memory prior to executing a program, some error messages were issued by AUGMENT when there was no error. This has been corrected by Dr. Crary in version 4K. Version 4K also corrects a problem of not recognizing DATA statements properly. This problem was noted on the DEC System 10 and IBM System 370, but only a "comment" type error message was produced so it was ignored.

3. Interval Arithmetic Package Implementa.ion

General Information

The interval arithmetic package for the 1108 [5] was received on tape and restored to disk on the System 370. A careful study of this package and 1108 manuals [15,16] revealed that an overwhelming number of machine dependent features are employed in the coding. These dependencies have been documented but are not included here since a more portable version of the package (INTERVAL II) is now available. Because the package would not convert easily to other machines, and since this project dealt with three distinct machines, it was decided that time would not be wasted in adapting this package.

Attention was turned to implementing the machine-dependent primitives required by INTERVAL II. The discussion that follows is concerned with the implementation of the arithmetic primitives which perform directed roundings. These were coded from Dr. Yohe's paper [4]. The source listing of the INTERVAL II primitives and constants are included in the appendicies for all three machines.

Making corrections to the INTERVAL II package deserves comment. The package is written using BPA and EXTENDED data types and therefore must be processed through AUGMENT. The resulting Fortran requires significant editing to tune the package. Thus, changes to INTERVAL II, such as corrections, can be made at either the AUGMENT input level or the AUGMENT output level. In view of the tuning required, it is recommended that changes apply to the AUGMENT output as well as input. This avoids having to reprocess through AUGMENT and retune the package.

Implementation on the DEC System 10

The five primitives are coded as one program with the entry points of BPAADD, BPASUB, BPAMUL, BPADIV, BPACEB. The Macro (assembly language) code was designed for communication with code generated by the DEC-10 F10 Fortran Compiler. Code compiled by the F40 Compiler will not interface correctly as the calling sequences are different. This is documented at the beginning of the primitive code.

The ACC value is assumed to be 28 in all cases of calls to BPACEB. There was no extensive information on error bounds for the DEC-10 thus double precision results are assumed correct to 28 bits or one more bit than single precision.

The primitives were coded directly from the paper "Roundings in Floating Point Arithmetic" by J. Michael Yohe. There are some minor problems that should be pointed out for anyone who would code from this paper directly. The occurrences of P+1 in lines 3, 4, 15, and 16 of table I (p. 580) should be replaced by P, as the stated constant is not the one that is exactly halfway between max and the real number whose exponent is EMAX+1 and fraction is B-1. There are several typographical errors which may slow down the coding procedure significantly. Fortunately only one such typo occurs in the algorithms portion. It is an obvious error in Step 6 of the rounding algorithm.

In implementing the algorithms one should check exponent overflow first when considering the rounding options applied to circle (O). If this is not considered first, one can easily implement a round toward zero when in fact the option dictates a round from zero.

The DEC-10 does not have an infinity representation. Thus positive infinity is just the largest positive number or

377777777777

in octal and

400000000001

for negative infinity.

The DEC-10 presents some interesting problems of manipulation as negative floating point numbers are in the form of 2's complement notation for the fraction and 1's complement notation for the exponent. The approach taken is to convert numbers to sign magnitude form for the algorithms and then convert back to DEC-10 representation upon exit from the primitives. Since the DEC-10 has no sign magnitude add instruction one must adjust to 2's complement form prior to and possibly after the addition.

A final obstacle that should be noted on the DEC-10 implementation is that of type checking on subprogram parameters. INTERVAL II represents intervals as REAL dimensioned by 2. In utilizing the package, an AUGMENT user must also declare the interval type as REAL dimensioned by 2 and not COMPLEX as on the System 370. It is not possible to suppress type checking under the F10 Compiler.

Implementation on the IBM System 370

Before implementing the primitives on the System 370, several problem areas had to be resolved. A decision had to be made with regard to the base of the machine. Although information is stored internally in binary, the exponent in floating point numbers represents a hexadecimal move on the hex point. An attempt was made at coding the algorithms of Yohe's paper using beta as 16, p equal to 6, and m equal to 3. The algorithms require m to be at least 3 and this meant that the A register had to carry 9 hex digits. Since a single 370 accumulator can only accommodate 8 hex digits, significant coding problems were continually encountered. The base was changed to 2, p to 24, and m set at 7. The algorithms were then coded successfully with the A and U registers as single 370 accumulators (contrary to the remark on page 579 of the paper).

This change to base 2 introduced some complexity into the packing and unpacking operations. The hex exponent was converted to binary, and a normalized hex number may not be a normalized binary number. Since an unnormalized number could be passed into the primitives, at the beginning of each algorithm a normalizing loop appears which shifts the fraction digits to the left and decrements the value of the exponent until normalization is achieved.

The individual algorithms presented little or no difficulty, after several problems encountered in the System 370 were worked out. The first of these was found in the multiplication algorithm. After a 32-bit multiply takes place, the second half of the 64-bit product contains a data bit in the sign bit portion. Accordingly, the result should be left shifted one bit to maintain the algorithm's assumption about the location of the binary point. In algorithm 4, division, the System 370 divide places a sign bit in the sign bit portion of the remainder. It is desirable to place a data bit in this position so that data bits in the AX register will be contiguous. The result must be left shifted one position and the sign of the remainder eliminated.

The first difficulty encountered in algorithm 5 was a misprint in the article. In Step 6 the instruction printed as "EA=EMIN**-2" should have read "EA=EMIN-2". Testing the rounding options for this algorithm was a problem, in that a number of different cases had to be considered for rounding option 4, round to the nearest machine number. At first only the two cases of this option which rounded toward zero were tested and a default to round away from zero would be taken if these were not found. However, the case which states that if the absolute value of the result is greater than or equal to MIN and the first digit of the second register is a zero implied rounding toward zero, without taking into consideration the size of the exponent. When the exponent is greater than MAX, a rounding away from zero is called for. So a test is made of the exponent's value preceeding the testing indicated in Step 6.

Also in algorithm 5, extra code was added at Step 10 since the System 370 would expect an exponent expressed to move hex points rather than bits. It is necessary to ensure that the exponent be a multiple of 4. Shifting the result and incrementing the exponent until this is achieved and dividing the exponent by 4 will generate the proper representation. Note that this causes the desired rounding to take place again if non-zero bits are shifted out.

The BPACEB algorithm required a 64-bit addition, which is not provided on the system 370. The code, therefore, had to make use of temporary storage locations to perform 32-bit additions, save the carry-out of the answer, and then add it to the upper 32-bits of one of the addends before continuing the addition. This accounts for the lengthiness of this code.

Both the checking and debugging of these routines were hampered by the fact that the System 370 accepts only batch jobs. There was no way to interactively debug, and much time was spent desk-checking the programs and using Fortran subroutines to print values at various points in execution. The cases in Table I were of great value in proving the validity of the primitives, but difficulty was found in producing correct test data with which to work. Often one would think that the routines were wrong when actually the test data was incorrect.

As a final note it is recommended that the System 370 primitives be extensively optimized before going into any production use. They can benefit from much tuning.

Implementation on the DEC PDP-11/70

In the PDP-11 a floating point number is stored in two consecutive 16 bit words. There is a 1 bit sign, 8 bit exponent, and a 23 bit fraction. Since all real numbers are assumed to be normalized the first bit of the fraction is not present in the floating point representation. This bit, called the hidden bit, plus the 23 fraction bits gives a 24 bit fraction. In the BPA routines the hidden bit is placed in the fraction when the number is unpacked. This makes $m=7$ (number of exponent bits) and $p=24$ (number of fraction bits) in the algorithms. The hidden bit is removed when the number is packed back into floating point format in Step 11 of Algorithm 5.

To implement the BPA algorithms on the PDP-11 three special routines had to be written; a 64 bit right shift, a 32 by 32 bit multiply, and a 64 by 32 bit divide. Since the PDP-11 is a 16 bit machine it takes two consecutive registers or words of memory to represent a floating point number. In the discussion that follows the A,X,U, and Y registers are actually two consecutive registers or words.

The 64 bit right shift routine takes the A and X registers as a single 64 bit register and shifts it right n places. To do this the A register is saved in Y then shifted right n places. The high order bit of the X register is placed into the low order bit of Y and then the X register is shifted right n places. The Y register is shifted left 31-n places to get the bits that were shifted out of A. These bits are then ORed into the X register.

A routine that multiplies two 32 bit numbers and produces a 64 bit product was written for BPAMUL. The A register is saved in Y and then cleared. Then U (the multiplier) is shifted right 1 bit at a time. If the bit shifted out is a one then Y is added to A and then AX (the product) is shifted right one bit. If the bit shifted out of U is a zero the Y register is not added into A, but AX is still shifted right one place. This continues until all 32 bits of U have been shifted out and tested.

For BPADIV a divide routine was written that divides a 64 bit dividend by a 32 bit divisor and produces a 32 bit quotient and a 32 bit remainder. The 64 bit dividend in the A and X registers is shifted left one place, putting a zero into the low order bit. The 32 bit divisor in U is subtracted from A. If the result is negative the divisor is added back in. If the result in A is positive the low order bit of X is changed to a 1. This sequence of shifting and subtracting is repeated 32 times. Then the A and X registers are exchanged so the remainder is in X and the quotient is in A.

Testing

Testing of the interval package on all three machines was carried out in a similar manner. However, debugging of the primitives was much simpler on the timesharing systems. The test driver supplied with the package was run successfully and produced satisfactory results on all three machines. Output from this test is included in the appendix.

Additional testing was made by executing Dr. Yohe's factorial problem [5] and Plat Map problem (illustrated at the conference in Vicksburg). The FFT benchmark program was also run as a test case. Output from running all three of these tests on all three machines is contained in the appendix. FFT run under normal arithmetic was used as a test comparator, and the output of the factorial problem as well as the Plat Map problem were available from the 1108 to use as comparators. All machines are producing satisfactory results for these problems.

As a guide to anyone else who implements the primitives using Dr. Yche's paper it should be pointed out that extensive testing of the primitives can be accomplished using Table I in the paper. Testing each one of the cases in that table to see if the proper results are produced generates a secure feeling that the primitives are coded correctly. A general driver program that exercises the primitives against this table is highly desirable.

4. Use of AUGMENT and the Interval Package

Using an Interval Data Type

Using AUGMENT for an Interval data type is a simple process and usually requires a minimimal number of changes to a user program. The number of changes that a programmer will need to make depends upon the description of the new data type given to AUGMENT. However, the description is limited to what the user has as library routines supporting the operations in the new data type. To define what is meant by the operations in the new data type, examine the addition operation. When doing an addition under standard data types, like integer or real, the machine has hardwired instructions which carry out the operation. When defining a data type not supported by the machine hardware, a software implementation of it must be made. The implementation is in the form of a subroutine or function which does the required operation or conversion. This is where the supporting package routines come in. All conversions, functions, and operations must be defined in the form of subroutines or functions where operands are passed to it and the results of the operation are returned.

Once all the operations, functions, and conversions needed by the new data type have been defined, information must be given to AUGMENT through a description deck. In the description, the symbol or name of each operation or function to be recognized and processed in the user program must be defined. An example would be '*' for the multiplication operation and 'SQRT' for the square root function. Next, the name of the supporting package subroutine or function which will do the same operation on the new data type must be defined. A specification of the types of the arguments passed, the priority of the operation, and the type of the argument returned must be defined. When all the operations and conversions have been defined, AUGMENT is ready to process the user's program for the new data type.

Before the writing of the description or the library routines, another aspect of the new data type must be defined, namely the data structure of the new data type. On the IBM 370, integers and single precision floating point numbers are stored in one word of memory. A complex number is stored as two words of memory with the first word containing the real part of the number and the second word containing the imaginary part while a double precision floating point number takes up two words of memory for the number by itself. When defining a new data type, not only must the operations be defined but the way a new data type is to be stored in memory must also be defined. In the case of Interval Arithmetic, each previously defined real variable will be stored as two real numbers which are the lower and upper endpoints of the Interval number. This can be done in many ways using various standard data type storage definitions to describe the new data type's storage requirements. The descriptions which could be used are REAL(2), COMPLEX, INTEGER(2), or DOUBLE PRECISION because each of these types are allocated two words of memory. AUGMENT defines the storage for each variable of the new data type to be a multiple of a defined standard type which the user supplies in the description to AUGMENT. For example, an INTERVAL array A(10) when defined by complex storage would be declared COMPLEX A(10) after being processed by AUGMENT. If an Interval was defined as REAL(2) then the declaration for the array A(10), after being processed by AUGMENT would be A(2,10). This definition maintains a column ordering hierarchy which Fortran uses. The problem of storage is simple as long as the data types are the same in the description of the data types and the way they are defined in the library routines. If they are not, then some compilers will generate code that gives errors because the parameter types of the calling program and the subroutine do not match at run time. Others may, at run time, make standard Fortran data type conversions on the parameters when they are passed or returned. This occurs with the DEC 10 Fortran 10 compiler. An example of this is found in the Fast Fourier Transformation problem that was implemented. An Interval was described to AUGMENT as COMPLEX but the supporting package used REAL(2) to represent an Interval. This caused the loss of the right endpoint upon return to the calling program. The IBM 370 Fortran compiler makes no check on the parameters. So as long as equal storage is defined in the main program and in the subprogram no error occurs.

To change a standard type variable to a new data type simply insert a statement into the user program which is similar to other type declaration statements in Fortran where the type name is followed by the list of variables to be given the type.

However, there are some small limitations when using AUGMENT to convert a program from a standard data type to a new data type. One of these limitations is that no tabs may appear in the source deck because AUGMENT does not process or recognize tabs in the source statements. An error results if one is encountered. Other limitations are that AUGMENT does not process EQUIVALENCE, DATA, READ, or WRITE statements. AUGMENT marks these statements in its output with "comment" messages:

```
C ===== EQUIVALENCE STATEMENTS ARE NOT PROCESSED BY AUGMENT =====
C ===== DATA STATEMENTS ARE NOT PROCESSED BY AUGMENT =====
C ===== NON STANDARD VARIABLE IN INPUT/OUTPUT LIST =====
```

This does not mean that these statements cannot be used. It means that AUGMENT does not make any conversion of these statements. Therefore an understanding of the data structure of the new data type is needed to ensure the correct alignment of storage and correct input and output of the new data type. For example, if A is an interval data type on the DEC-10,

```
      WRITE (6,50) I, A(I)
   50 FORMAT (1X,I4,5X,E14.7)
```

would have to be recoded by the programmer as

```
      WRITE (6,250) I, A(1,I), A(2,I)
  250 FORMAT (1X,I4,5X,1H(,E14.7,1H,,E14.6,1H))
```

To input or output a variable with an interval data type will mean changing the I/O statement to read both endpoints or write both endpoints. On the I/O statement, a list would have to be made of the variables using a subscript value for variables that are scalar if the definition of an interval was REAL(2). If the definition of an interval was complex then Fortran would be able to take care of the input and output of both endpoints with just the variable name. Either way, the format specifications would have to be changed to allow for an extra value to be read or printed for each variable in the variable list. With a READ statement the input data would have to be reformatted and changed so as to represent an interval number.

The programmer must be keenly aware that to change a standard variable in an EQUIVALENCE statement to a new data type may mean changing the array size of the variable it is equivalenced to. For other statements, AUGMENT handles the extra subscript and it becomes transparent to the user. Changes to other statements may have to be made if the supporting package routines are not available for the particular data type desired. As an example, the Fast Fourier Transformation (FFT) generates a set of complex coefficients for a polynomial. Since there was an interval package library for a REAL interval number but not for a COMPLEX INTERVAL number, the complex number was simulated by declaring the old COMPLEX array as a REAL array dimensioned (2,N) were N was the old array size. This, by itself, meant changing some of the code to allow for the subscripts and change the logic from storing one complex number to storing two real numbers. For example it is now the programmer's responsibility to keep track of the real and imaginary parts separately. The FFT statement

```
        WP(L)=CMPLX(X,Y)
```

was recoded into interval as

```
        WP(1,L)=X
        WP(2,L)=Y
```

This change to the standard non-interval program would still maintain the logic of the algorithm, provided the change is made correctly. Once this correction was made, the change from the standard types to interval were transparent except for the I/O statements and EQUIVALENCE statements. The reason this special change was made instead of writing another set of library routines to handle a complex interval was that the time needed to write the new library routines was much greater than the time needed to modify the logic of the one FFT program which was to be converted.

Two other points on using AUGMENT are pertinent. One, the implied comparison to zero used by the three branch IF statement:

```
IF ( arithmetic expresion ) label1,label2,label3
```

must be changed to the logical if statement (more than one may be needed):

```
IF ( arithmetic expression .EQ. 0 ) GO TO label2
```

This case is flagged by AUGMENT and is easy to spot and correct.

The second point involves the manner in which AUGMENT generates type declaration statements. To illustrate, consider an INTEGER array that is dimensioned by an INTEGER variable in a subprogram. The AUGMENT declaration would be flagged by the IBM Fortran compiler with an "order" diagnostic. That is,

```
C          ----- GLOBAL VARIABLES -----
      INTEGER LABEL(NRARG), NRARG
                                  $
***** 01) (CODE) ORDER
```

Two easy solutions are to dimension the array with a constant such as one (1), or to rename the scalar INTEGER variable as KNARG for example. This second solution is valid because AUGMENT produces an alphabetical list of the variables in the generated declaration.

On a final note, when a run time error occurs, it is usually detected and printed in INTRAP. The reader is referred to the appendix for an example of this. The output, however, is difficult to relate back to the AUGMENT input for a program of significant size. It is believed that AUGMENT's output will have to be used in debugging. This means that one should be familiar with the individual routines in INTERVAL II. It is useful to have a list of routine names, a brief description of their function and parameters. The reader is referred to [5, Table I, pp. 78-79] for an example list for the 1108 package.

5. Recommendations and Conclusions

The AUGMENT preprocessor and the INTERVAL II package are working products. Moreover, they are portable to a great extent with the exception of the machine-dependent primitives. However, the coding of the primitives is well defined. Specific recommendations include the following:

1. AUGMENT can be implemented on a small storage machine with suitable restructuring and modification. Much code can be eliminated that would significantly reduce the core requirements.

2. It is believed that AUGMENT and INTERVAL II machine dependent primitives can be made machine independent to a great extent.

3. The interval I/O package [6] is probably a useful tool to use in conjunction with INTERVAL II in light of the confidence a programmer has in the Fortran I/O system. This package should be implemented on the System 370, DEC-10, and PDP-11/70.

4. As noted earlier a general driver program that exercises the INTERVAL II primitives is highly desirable.

PART II: EVALUATION OF INTERVAL ARITHMETIC ON THE IBM 370, DEC 10, AND DEC PDP-11/70 SYSTEMS

6. Introduction

Organization of this Part

The objective of Phase II of the project has been to evaluate interval arithmetic on the DEC System 10, IBM System 370 and DEC PDP-11/70 computers. Interval arithmetic is a useful tool in the analysis of algorithms and hardware [19]. The quality of the above systems in terms of accuracy is highlighted by interval arithmetic results on the benchmark programs. Inconsistencies and weaknesses in hardware design are most apparent.

With the above in mind, this Part has been organized as follows. Roundoff and truncation error as pointed out by running Dr. Yohe's benchmark programs is discussed first. Results discussed here relate to accuracy information supplied in later sections. Next, the results of the benchmark testing are discussed. Following, a comparison of the three machines is given in terms of the benchmark program testing. This Part ends with recommendations and conclusions.

Summary of Phase II Activities

Important Observations -

* The DEC System 10 is superior to other machines observed in both interval arithmetic and normal floating point arithmetic.

* The hexadecimal point move employed by the IBM System 370 in its floating point representation severely hampers the machine's ability to generate accurate results in single precision arithmetic. This is reflected in the interval arithmetic results.

* Using interval arithmetic is made simple because of the AUGMENT preprocessor.

* Interval arithmetic, including AUGMENT use, will increase processing time by as much as a factor of 50. This is not so on the PDP-11/70 because the time needed for normal arithmetic operations is large anyway.

Problems -

* With the increased storage required for interval arithmetic (a factor of 2) and considering the PDP-11/70 storage limitations, it is doubtful that problems of significant size can be solved on this machine.

* Although testing of the INTERVAL II package has been extensive, a high degree of confidence in the validity of the results has not yet been established. This should change with more extensive use of the system.

* Use of AUGMENT output in debugging is mandatory and is complicated by not having a convenient method of tying the AUGMENT input to its output.

* Interpretation of interval widths is difficult when the situation is not obvious. Guidelines based on practical experience are much needed in this area.

Addendum to Part I

Further use of AUGMENT [1] since the writing of Part I has revealed two additional comments. One pertains to using AUGMENT with an interval data type. On the DEC System 10, type INTERVAL is mapped to REAL dimensioned by two, and is mapped to COMPLEX on the IBM System 370. AUGMENT on the System 10 incorrectly handles the following declarations:

```
INTERVAL  R1,R2,R1RPM
REAL  RR1,RR2,RR1RPM
```

It does so by failing to produce a declaration of the form:

```
REAL  R1(2),R2(2),R1RPM(2)
```

However, on the System 370, AUGMENT generates correct results. That is:

```
COMPLEX  R1,R2,R1RPM
REAL  RR1,RR2,RR1RPM
```

The other remark concerning AUGMENT use applies to debugging a program processed under AUGMENT. AUGMENT maps operations on the new data type to calls in the supporting package. It does not, however, relate the output generated back to the original source. Since the mapping of source to output is not on a statement per statement basis, debugging is an intolerable task. For example, PRINT statements to check intermediate results between calls on the supporting package cannot be inserted in the original source program. Thus, the programmer is forced to use AUGMENT's output. This notwithstanding, and unavoidable, it would be desirable to have an optional listing feature such that when turned on, a source statement would be printed followed by the sequence of statements produced from it. The programmer would still be required to work from AUGMENT's output file. However, a listing of this nature would simplify its use.

Part I of this report fails to mention that the DEC System 10 has no double precision Arcsin and Arccos (DARSIN and DARCOS) routines. The single precision routines ASIN and ACOS were used in implementing INTERVAL II.

Further testing of the INTERVAL II package on the PDP-11/70 has revealed an error in the INTERVAL II constants THPI, EXPMNA, and FRACBD. The new values for these constants on the PDP-11/70 are as follows:

```
THPI   /"31371641026/
EXPMNA  /-88.028/
FRACBD  /"000000046000/
```

The latter constant invalidated the arithmetic results of the FFT benchmark using interval arithmetic on this machine. A revised version is available at the Waterways Experiment Station (WES).

Fortran under the PDP-11/70 will not permit the printing of REAL variables using octal output format. The same holds for input using the READ or DATA statements. This has been a persistent problem not only in implementing the INTERVAL II package and its primitives, but in the testing of the benchmark programs.

In relation to this problem, values of type REAL must be EQUIVALENCEd to INTEGER variables for input/output in octal. However, the 11/70 stores 32 bit integers in memory with the low order 16 bits above the high order 16 bits. This ordering is preserved in octal input/output. Hence, real values must be interpreted with the sign and exponent flanked on either side by fraction digits.

7. Roundoff and Truncation Error

Interval arithmetic can highlight roundoff and truncation error in a computer. This has been reported on by Dr. Yohe [19].

It is well known that the order in which the terms of a summation are added can affect the accuracy of the result. When the sizes of the terms vary significantly, interval arithmetic will give an excellent idea of the effect of roundoff error. Computer output from all three systems was produced for the sum:

$$y = \sum_{i=0}^{128} \left(\frac{1}{x}\right)^i$$

for several values of x ranging from -13 to 11. In summing forward (largest terms to smallest), the interval sums on the DEC System 10 are as wide as $.2_8 \times 8^{-6}$. Summing backwards the interval sums are no wider than $.2_8 \times 8^{-8}$. This difference in the interval widths underlines the effect the order of summation can have on accuracy due to roundoff error. This is even more true when considering the System 370 where the forward sum interval widths were as high as $.8_{16} \times 16^{-4}$ and the backward sum widths were up to $.4_{16} \times 16^{-5}$.

Throughout the remainder of this report the reader will note significant differences between interval widths on the System 370 and the System 10 (or PDP-11/70). This discrepancy is directly attributable to the hexadecimal point move employed in floating point representation on the System 370.

For purposes of illustrating this point assume that we have two machines, both capable of representing four (4) bit fractions. Furthermore, suppose that the first machine's exponent value implies a binary digit move on the fraction, and the second machine's exponent value implies a hex digit move on the fraction. After an arithmetic operation, but before rounding occurs, assume a result of $.00011_2$ is developed. In the terminology of [4] for the first machine we have $e_A=-3$, the fraction portion of the A register as .1100, and the X indicator off. For the second machine, $e_A=0$, the fraction portion of the A register as .0001 (or $.1_{16}$) and the rounding indicator X on.

Assume an upward directed rounding, and examine the results of rounding. For the first machine the X indicator is off implying the machine can exactly represent the result. Thus, rounding is not done and has no effect. For the second machine, however, the indicator X is on, and rounding would occur yielding a result of $.001_2$. A downward directed rounding on the first machine would yield the same answer, but $.0001_2$ on the second machine. From this analysis it can be observed why the System 370 produces wider intervals than the DEC System 10 and the PDP-11/70.

This fact also influences results in normal floating point arithmetic. In general one can expect the System 370 to produce less accurate results than machines which have a bit move on the fraction.

On the preceeding summation problem, the forward and backward sums were also produced using normal arithmetic. The widest discrepancy between the two sums on the System 10 was $.3_8 \times 8^{-8}$, and $.B_{16} \times 16^{-5}$ on the System 370. Again the hex point move on the fraction came into play. In summary, error control on the DEC System 10 and PDP-11/70 is tighter than that on the System 370.

To see how serious roundoff error can affect a computation we again turn to a benchmark program supplied by Dr. Yohe [19]. In computing the roots of a quadratic equation, it is well known that blind application of the quadratic formula can yield very inaccurate results. Interval arithmetic can show just <u>how bad</u> the error can be in the case where 4ac is small compared to b^2.

Runs were made applying the quadratic formula on the three systems in question. The smaller root was computed using both the quadratic formula and by dividing the larger root into c/a. The roots were also computed using normal arithmetic in addition to interval arithmetic.

Although several sets of values for a, b, and c were used, for purposes of illustration between the machines, we will consider the case where:

$$a = .186264515 \times 10^{-8}$$
$$b = 1.$$
$$c = 1.$$

Floating point arithmetic on the System 370 produced the smaller root value as 120.0 using the quadratic formula. Note, that the System 10 yielded a value of -2 for the same case. It turns out that both values are nonsense, but the magnitude of the error on the System 370 is appalling. The result does not even have the correct sign. The System 10 value is off only by a factor of 2, and at least it has the correct sign.

The interval results are even more interesting. The smaller root computed via the quadratic formula and interval arithmetic yielded [-16.,256.] on the System 370, and [-4.,0.] on the System 10. These intervals are uselessly wide, but in regard to algorithm analysis, they demonstrate dramatically how unstable this algorithm is, especially on the System 370. The division algorithm is very stable considering the values processed. On the System 10, the left and right endpoints are consecutive machine numbers--[577377777777_8,576400000000_8], which is the best possible interval that can result from an inexact computation. Even the System 370 produced an acceptably narrow interval--[$C1100001_{16}$,$C0FFFFF0_{16}$].

In his paper [19], Dr. Yohe also illustrates the effects of a design flaw in the Univac 1110. That computer truncates the addends (presumably) to 27 bits before performing an add or subtract. The illustration is made by using a contrived constant of 2^{-27} and subtracting it from 1.0 twice. On the 1110, the truncation of 2^{-27} down to 27 fraction bits (after binary point alignment) results in subtracting zero from 1.0 twice. This difference defined the first term of a sequence x_1, and further terms were computed using $x_i=x_{i-1}^2$ for $i=2,3,\ldots,40$. Finally, to observe the effect of the truncation error, the x_i's were summed from the left and the right. The results of running this same analysis on the System 10, System 370 and PDP-11/70 are available at WES.

Analysis of the output yields some interesting results about floating point arithmetic on the three computers. To begin, compare the System 10 with the Univac 1110 [19]. Both machines have 36 bit words with an internal floating point representation consisting of a sign bit, an eight bit exponent, and a twenty-seven bit fraction. Hence, the constant 2^{-27} that was used on the 1110 is valid for illustrating the same point on the System 10. However, a study of Appendix C shows that the System 10 produced a value of approximately 25.66725_{10} instead of 40.0_{10} as the 1110. It has been pointed out that the 1110 truncates before adding or subtracting and this was the design flaw. However, on the System 10 a double length accumulator is used in arithmetic, and so the subtraction of 2^{-27} from 1.0 does not yield 1.0. Note, also, that the System 10 normal arithmetic value fell within the interval arithmetic value for each case. Interval arithmetic on the 1110, however, produced an answer far better than the 1110 floating point arithmetic.

The critical point here is that even though these two machines have essentially the same internal representation, normal arithmetic results on the System 10 can be expected to be more accurate since greater precision is used in actually performing the arithmetic. Both machines' interval packages can be expected to perform the same (disregarding extended precision functions that are used).

This example permits us to gain more insight into System 370 arithmetic. Specifically, it reveals that 40.0 was produced as the sum in normal arithmetic, and unexpectedly, the interval analysis also yielded an unacceptably wide interval-- $[22.66682_{10}, 40.0_{10}]$. Since the System 370 only has 24 fraction bits, the use of the constant 2^{-27} may be questioned. However, in normal arithmetic a four bit guard digit participates in addition to the 24 fraction bits. One would think that the 28 bits combined would accomodate the value 2^{-27} subtracted from one. They would if the System 370 exponent value moved a binary digit in the fraction! The hex digit move on the fraction once again causes unacceptable results to be produced. Recall that a normalized one would appear as $.100000_{16}$ with an exponent of 1. The value 2^{-27} would appear as $.100000_{16}$ with an exponent of -6. Alignment of the hex points before subtracting causes the non-zero digit of 2^{-27} to be right shifted. It is clearly shifted out of the guard digit. The value of having the guard digit is lost. Notice that the three zero bits leading the fraction of 1.0 are not even used in the arithmetic! In fairness, however, it is possible to devise a constant such that the hex digit move on the fraction does not override the utility of the 4 bit guard digit. The System 370 designers at least had it over the Univac 1110 designers in recognizing the need for some additional bits of precision. But they more than made up for this in designing a hex digit move on the fraction.

On the PDP-11/70 at least one additional bit of precision is used in arithmetic and a constant of 2^{-24} will not produce the same effect as the corresponding constant did on the 1110. However, a constant of 2^{-25} causes normal arithmetic to produce the value of 40., and interval arithmetic to produce the value of 40.0 on the right endpoint. The value of 40.0 on the right endpoint is explainable due to the upward directed rounding. On normal arithmetic, however, the 40.0 could only result by one of two things. Either there is only one additional bit of precision used, or if more than one is used, the CPU performs a rounding operation. This operation is available on the DEC System 10.

8. Benchmark Testing

Recommendations for error and sensitivity analysis of algorithms can be found in [19]. Primarily, error in an algorithm occurs due to data sensitivity. A given algorithm can be pronounced reliable if it produces acceptably narrow intervals for several representative sets of data. Subsequently, however, there is always the fear that there exists data for which the algorithm is sensitive to. Such being the case, in a production environment where a particular situation dictates that accuracy is of critical importance, interval arithmetic should be used to confirm the validity of the results. This is especially true if the reliability of the algorithm has not been established with analogous data.

As pointed out in [19], if an algorithm generates unacceptably wide intervals for a given set of data, that algorithm should be examined for data sensitive operations. These operations can be located by printing intermediate results to discover where accuracy is lost.

Once such perturbations are found, the programmer may consider one of several methods in correcting the problem [19]:

1. Is there an equivalent, but more stable algorithm which can be used?
2. Is there a critical summation which could be made more accurate by judicious choice of the order of summation?
3. Can the critical portion be rewritten using higher precision arithmetic in such a way as to improve accuracy?

There are several handicaps in using interval arithmetic. Widespread availability of the tool is nonexistent. However, projects such as this one will help to alleviate this problem. One can at least expect that the amount of main storage required to run the computation will double. This is a direct result of converting REAL numbers to INTERVAL numbers, the size of the INTERVAL II package itself (59K on the System 370), and the overhead of AUGMENT generated calls on the package. For example, the FFT benchmark run under normal arithmetic required 128K on the System 370. Under interval arithmetic, 254K was required to run the program. The storage increase is observed at approximately 100%, or double the amount. On the 11/70, the FFT benchmark would not fit in storage for 512 input points under interval arithmetic. It would, however, under normal arithmetic. This is why FFT with only 256 input points was run.

One of the more obvious penalties of using interval arithmetic is that of increased CPU time. With the absence of hardware instructions that perform the rounding operations described in [4], software must be written to simulate them. Every interval arithmetic operation performed in INTERVAL II incurs the overhead in CPU time required to simulate the operation in software. To get some idea of how dramatic this increase can be examine Table 1. Using approximate figures, the System 10 increase is a factor of 27; on the System 370, a factor of 50, and a factor of 2 on the PDP-11/70.

Table 1

CPU* Time (in seconds) for the FFT Benchmark Program

	Normal Arithmetic	Interval Arithmetic
System 10	3.97	105.9
System 370	10.06	471.2
PDP-11/70	32.60**	72.4**

* The time given includes that required for compiling, linking/loading, and execution. For interval arithmetic, AUGMENT CPU time is also included.

**FFT was processed for only 256 input points on the 11/70 due to storage limitations. The other systems processed FFT for 4096 input points.

This time on the 11/70 is shaded by the inordinate amount of time it takes to task build. Ignoring this time, the increase in CPU time of interval arithmetic over normal arithmetic is a factor of 10. This relatively small time factor on the 11/70 is an indication of how slow normal arithmetic is on that hardware anyway. This is very important if one considers the BPA arithmetic primitives on this machine (multiply, divide, and shift instructions had to be software simulated). Tables 2 and 3 give a breakdown of the figures in Table 1. The dramatic increase in the System 370 CPU time is due in part to the inefficiency of the BPA primitives noted in Part I. However, there are many instances where the instruction set on the System 370 falls short in comparison to the System 10. For example, handling a simple exchange operation on the System 370 takes three instructions and only one instruction on the System 10.

As Dr. Yohe points out it is no longer regarded as an "unpardonable sin" to have software inefficiencies, especially considering the rate at which hardware prices are falling, and " . . . the extra cost may seem small when balanced against the possible failure of a structure [19]."

Analysis of the arithmetic results of running the FFT benchmark program yielded the following observations. Since only one set of input data was processed, it is not advisable to pronounce the algorithm reliable as defined earlier. The input data points range in $-1 \leq x \leq 1$ with a magnitude of

Table 2

Individual Program CPU Times (in seconds) for FFT under Normal Arithmetic

	System 10	System 370	PDP-11/70
Fortran Compiler	.85	3.10	5.15
Linker/Loader	1.38	1.00	25.42
FFT Processing	.74	5.96	2.04
Total	3.97	10.06	32.6

Table 3

Individual Program CPU Times (in seconds) for FFT under Interval Arithmetic

	System 10	System 370	PDP-11/70
Augment Processing	16.77	22.19	16.77*
Fortran Compiler	1.48	5.85	7.41
Linker/Loader	1.96	3.58	27.01
FFT Processing	85.69	439.48	21.21
Total	105.9	471.2	72.4

* Augment on the DEC System 10 was used as a host processor. See Part I.

10^{-1} on FFT under normal arithmetic. The method of computing the input data points using interval arithmetic produced intervals no wider than $.001219276_{10}$ on the System 370 and $.0001220703_{10}$ on the System 10. FFT under normal arithmetic generated complex numbers with a modulus of magnitude ranging from 10^{-5} to 10^{-7} on the System 370, and 10^{-6} to 10^{-9} on the DEC System 10. Under interval arithmetic, the widths of the interval valued moduli were no wider than $.001061307_{10}$ on the System 370 and no wider than $.000677995_{10}$ on the System 10. Considering the magnitudes of moduli under normal arithmetic, the widths of the intervals are not too encouraging. However, the widths reported on above are the widest that occurred. By far the majority of the output points had modulus interval widths of magnitude 10^{-3} on the System 370 and 10^{-4} on the System 10. Also, the output interval widths seem reasonable considering the widths of the input intervals. This is reassuring and would indicate that this particular FFT algorithm is not sensitive to the sample test data.

Earlier, the term "acceptably narrow" intervals was used in reference to determining data sensitivity of a given computation. It can be observed from the discussion above that exactly what is acceptable and what is unacceptable is not clear cut. There exists a grey area where indecision may occur. After experience is gained in using the tool, it might be judicious to determine beforehand what an acceptable interval width would be, and then run the computation under interval arithmetic.

Dr. Yohe supplied benchmark programs to test the performance of the SIN, COS, and TAN routines. The results of running this benchmark are available at WES, as are the results of running a second benchmark program supplied by Dr. Yohe. These benchmarks tested the mathematical functions using INTRAP for output. The output was examined for consistency. As expected, the DEC System 10 results were more accurate than the System 370. However, both systems generate results that are agreeable in the first seven significant digits. The input intervals staircase up to a certain point, and then staircase down with the endpoints of adjacent intervals overlapping. In a similar manner, the output intervals slightly overlap on adjacent endpoints.

On the testing of EXP, the input interval [-1. x 10^3, 0.] generated distinctly different results on the System 10 and System 370. The former produced an interval of [0,1], and the latter produced an interval of [0, min]. These results are explainable in terms of the interval arithmetic constants for the System 370. The constant EXPMNA is valued at min on the System 370 when it should be valued at -AE.AC4E$_{16}$. In constructing these constants it was not understood that negative numbers were to be included in the meaning of the term "smallest." The error is easy to correct.

9. Comparison of the Machines

As pointed out earlier, the hex point move on the fraction causes results to be less accurate on the System 370. The System 10 is more accurate than the 11/70 considering the difference in the number of fraction bits in their internal floating point representations. This pertains to both normal floating point arithmetic and interval arithmetic. Aside from accuracy comparisons, it is important to consider timing relationships between the machines due to the extra cost involved in using interval arithmetic.

The validity of timing comparisons in this section is dubious considering the discrepancy that exists in the efficiency of the BPA primitives. It should be noted that the BPA primitives on the 1110 are very efficient compared to those on these three machines. To observe this, one only has to examine the instructions available on the 1110 used to implement the primitives. This should affect timing comparisons with the 1110.

Table 4 provides timing information for eight benchmark programs that were run on all three machines. The table contains for each machine and each benchmark four CPU processing times (in seconds)--AUGMENT processing, Fortran compiler processing, linker or loader processing, and the benchmark processing time (under interval arithmetic). AUGMENT processing time on the 11/70 is the same as for the System 10 since that machine

Table 4

CPU Time (in seconds) for UT-Arlington Benchmark Programs

		Augment Processing	Fortran Compiler	Linker/ Loader	Program Processing
INTERVAL II Test driver	IBM 370	—	4.70	2.69	2.58
	DEC 10	—	.85	1.38	.74
	PDP 11/70	—	6.11	40.44	1.45
Plat Map Problem	IBM 370	7.66	2.15	2.84	.31
	DEC 10	7.51	.89	1.38	.08
	PDP 11/70	7.51	1.46	30.01	.12
Interval Factorial	IBM 370	5.14	.83	2.64	3.26
	DEC 10	5.93	.46	1.38	1.31
	PDP 11/70	5.93	.45	30.23	2.11
$\sum_{i=0}^{128} \left(\frac{1}{x}\right)^i$	IBM 370	5.75	1.11	2.54	10.68
	DEC 10	5.93	.31	1.38	2.47
	PDP 11/70	5.93	1.21	26.46	10.55
$\sum_{i=2}^{40} x_i=x_{i-1}^2$	IBM 370	6.63	1.23	2.90	.65
	DEC 10	6.12	.36	1.36	.19
	PDP 11/70	6.12	1.30	25.00	.33
Roots of Quadratic	IBM 370	7.52	1.90	3.07	2.25
	DEC 10	7.42	.58	1.40	.57
	PDP 11/70	7.42	2.20	26.04	2.04
Mathematical Programs	IBM 370	7.40	2.23	2.87	18.18
	DEC 10	6.56	.55	1.38	9.48
	PDP 11/70	6.56	3.05	25.24	20.02
SIN, COS and TAN	IBM 370	6.97	1.66	2.98	1.92
	DEC 10	6.44	.40	1.53	.77
	PDP 11/70	6.44	1.56	24.4	1.51
Total	IBM 370	47.07	15.81	22.53	39.83
	DEC 10	45.91	4.4	11.19	15.61
	PDP 11/70	45.91	17.34	227.82	38.13

was used as a host processor. The table summarizes the four processing times via a total for each system.

Examining AUGMENT time, it can be observed that there is not an appreciable difference between the System 10 and the System 370. On all other figures, however, the System 10 CPU time is significantly less. The System 10 compiler, linking/loading operation, and benchmark time are all at least half that of the System 370 and the PDP 11/70. The KL10 processor and software system are undoubtedly faster than the System 370/155 operating under OS/MVT. The high linker/loader time on the 11/70 is probably due to the fact that only one disk is configured into the system.

10. Recommendations and Conclusions

Using interval arithmetic as prescribed in the INTERVAL II package is simple because of the AUGMENT preprocessor. Computer runs illustrate that interval arithmetic can show the instability of an algorithm for a given set of data. Thoughtful use of PRINT statements and interval arithmetic can reveal sensitive parts of an algorithm. On the other hand, use of interval arithmetic can establish a high level of confidence in an algorithm for a given set of data. Proper testing with representative data sets can establish algorithm reliability. There is a peralty of increased CPU time and main storage requirements in using interval arithmetic.

The DEC System 10 appears to be the superior machine taking all factors into consideration (ie., time, storage, accuracy). The System 370 appears to be the worst machine considering the accuracy of results because of the hex point move on the fraction.

Definite problem areas are: storage limitations of the 11/70; confidence in the results of the interval package; and interpretations of the interval widths when the decision is not clear cut.

Specific areas for further work besides the obvious ones of continued benchmark testing and more experience in using the tool include the following:

1. Some computer-assisted training modules should be prepared to provide ready instruction on how to use AUGMENT and the INTERVAL package.

2. A validation package should be constructed that can be used to verify the correctness of the INTERVAL II package with its primitives.

3. Requirements exist for a double precision interval arithmetic package. These requirements should be considered to determine whether or not to implement such a package.

4. Similar to item 3, except the requirements are for a complex interval type.

PART III: EVALUATION OF BENCHMARK ALGORITHMS USING INTERVAL ARITHMETIC

21. Introduction

This work is a continuation of the efforts described in Parts I and II. WES benchmark programs GAUSE, BANSOL, SESOL, and SPLINE have been processed under normal arithmetic and interval arithmetic on three systems--the DEC System 10, IBM System 370, and DEC PDP-11/70. Here, we report on accuracy, storage requirements, and timing of these benchmarks and make recommendations and conclusions.

Summary Remarks

* Interval arithmetic should be used (as any tool would be) where accuracy is of critical importance.

* Because of the known numerical stability of Gaussian procedures in linear equation solving, the use of interval arithmetic is not recommended in the case of routines GAUSE, BANSOL, and SESOL.

* If accuracy is of critical importance, the back substitution portion of Gaussian elimination should be considered data sensitive because of the summations involved.

* Interval arithmetic is expensive to use in terms of computer time, main storage, and personnel time spent in error analysis.

* Some reasonable means of estimating the cost of using interval arithmetic in a given situation should be developed. These costs would be important in the decision process of determining whether or not interval arithmetic would be worth the effort or not.

* Techniques of accuracy extension on short word length machines should be examined, in particular the extended precision package discussed in [20].

12. Benchmark Accuracy

The approach taken in accuracy determination was to process the given benchmarks in normal arithmetic and then under interval arithmetic. The interval results were checked to insure they contained the normal arithmetic results. The widest interval widths were computed as well as the average width for each benchmark. These results are given in Table 5.

Table 5

WES Benchmark Accuracy

Benchmark	Result Range	Widest Interval	Average Width
SPLINE			
S/370	$10^0 - 10^2$	0.1062012E-01	0.4344657E-02
S/10	$10^0 - 10^2$	0.1118898E-02	0.4972159E-03
11/70	$10^0 - 10^2$	0.3030777E-02	0.1414956E-02
GAUSE (100 x 100 set of data)			
S/370	$10^{-6} - 10^{-1}$	0.6205976E-02	0.4425270E-03
S/10	$10^{-6} - 10^{-1}$	0.3089159E-03	0.2275300E-04

The question then arises as to whether or not the interval widths are acceptably narrow or not. Considering the numerical stability of Gaussian procedures, and the results ranges, the intervals are probably acceptable. The interval programs used were natural extensions of the normal arithmetic programs. No effort was made to eliminate dependencies that occur in the interval package. Otherwise, the interval widths would even be smaller. Without a specific situation to consider, it is hard to say for sure whether or not the widths are acceptable.

Because of the known numerical stability of Gaussian procedures, we recommend that interval arithmetic not be used in the case of these benchmarks. However, interval arithmetic should be used even in this case if accuracy is of critical importance. This is because the order in which the terms of a summation are added can affect the accuracy of a result, and the back substitution process of Gaussian elimination can be data sensitive. This is a possible failure point within these benchmarks.

One other remark on accuracy is important. The 100 x 100 set of data processed through the Gaussian procedures generated the interval widths given in Table 5. However, if one compared the solutions, they were distinctly different! How can this discrepancy be explained considering that the interval widths are acceptably narrow? The answer lies in the fact that an interval input/output package was not used in reading in the 100 x 100 data. Since the inputs were not properly bounded, we essentially solved separate systems on the different machines. This example highlights the importance of having an interval I/O package.

Benchmark Timing

Interval arithmetic is expensive to use in terms of CPU time. This is because every interval operation incurs the overhead in CPU time required to simulate the operation in software. To see how significant these times are, consider the figures in Table 6. For each benchmark, on each system, CPU times are given for each program under both normal arithmetic and interval arithmetic. Also given in Table 6 are factors of increase in CPU time in going

Table 6

Timing (in seconds) of WES Benchmark Programs

Benchmark/ System	Augment	Compiler	Loader	Program	Total	Increase
SPLINE						
S/370						
Normal	-	1.72	0.95	0.16	2.83	7
Interval	9.95	2.82	3.02	2.72	18.51	
S/10						
Normal	-	0.48	0.33	0.08	0.89	12
Interval	8.85	0.88	1.43	0.60	11.49	
11/70						
Normal	-	2.00	12.40	0.05	14.45	3
Interval	8.85	3.15	25.41	2.16	39.30	
SESOL						
S/370						
Normal	-	8.54	0.98	2.84	12.36	5
Interval	44.12	13.14	3.77	3.87	64.9	
S/10						
Normal	-	2.25	0.45	0.53	3.23	11
Interval	29.83	3.25	1.58	0.70	35.36	
11/70						
Normal	-	14.3	-	0.13	14.43	
Interval	29.83	18.23	-	(Would not fit in storage)		
GAUSE						
S/370						
Normal	-	3.92	1.01	0.39	5.32	6
Interval	22.25	5.81	3.14	1.04	32.24	
S/10						
Normal	-	1.08	0.40	0.13	1.61	13
Interval	17.33	1.95	1.61	0.24	21.13	
11/70						
Normal	-	6.40	13.51	0.15	20.06	2
Interval	17.33	10.26	20.16	0.56	48.31	
GAUSE (100 x 100)						
S/370						
Normal	-	3.68	1.00	14.82	19.50	79
Interval	22.09	5.77	3.12	25:08.16	25:39.14	
S/10						
Normal	-	1.03	0.38	8.45	9.86	55
Interval	17.03	1.60	1.53	8:45.00	9:05.16	

from normal arithmetic to interval arithmetic. Using approximate figures, the DEC System 10 increases by a factor of 12, the System 370 by a factor of 6, and the PDP-11/70 by a factor of 3.

Benchmark Storage

Another handicap in using interval arithmetic is that one can expect the amount of main storage required to double. Table 7 contains the storage requirements for the benchmarks on the System 370 and PDP-11/70.

Table 7

Storage Requirements of WES Benchmarks

	SPLINE		SESOL		GAUSE		GAUSE (100 x 100 Set of Data)
	S/370	11/70	S/370	11/70	S/370	11/70	S/370
Normal	30K	10K	52K	16K	36K	11K	74K
Interval	100K	22K	128K	>32K	108K	25K	224K
Increase	3	2	2	2	3	2	3

13. Conclusions and Recommendations

A problem area that remains is the lack of some quantitative means of determining the costs of using interval arithmetic on a given system. From the results of this Part and those in Parts I and II, one can estimate CPU time and main storage requirements for interval arithmetic if those figures are known for normal arithmetic. However, the costs of writing other than a natural interval extension program should be estimated. This would require programmer time in the elimination of interval package dependencies. Also, numerical analysts' time should be estimated in determining roundoff error and trunction error. This would leave inherent error as the essential error source, and interval widths could be appropriately evaluated.

Techniques of accuracy extension on short word length machines such as [[illegible] should also be considered.

REFERENCES

[1] Crary, F. D., "The AUGMENT Precompiler; I: User Information," The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report No. 1469, Dec 1974.

[2] __________, "The AUGMENT Precompiler; II: Technical Documentation," The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report No. 1470, Oct 1975.

[3] __________, "Guide for AUGMENT Implementation," The University of Wisconsin-Madison, Mathematics Research Center, Mar 1976.

[4] Yohe, J. M., "Roundings in Floating-Point Arithmetic," IEEE Trans. on Computers, Vol C-22, No. 6, Jun 1973, pp. 577-586.

[5] Ladner, T. D. and Yohe, J. M., "An Interval Arithmetic Package for the UNIVAC 1108," University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report No. 1055, May 1970.

[6] Binstock, W., Hawkes, J., and Hsu, N. T., "An Interval Input/Output Package for the UNIVAC 1108," University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report No. 1212, Sep 1973.

[7] Yohe, J. M., "Guide to Implementation of the Interval II Package on Other Hardware," University of Wisconsin-Madison, Mathematics Research Center, Jun 1976.

[8] DEC PDP-11/70 IAS User's Guide. (Order No. DEC-11-OIUGA-A-D.)

[9] DEC PDP-11/70 FORTRAN IV-Plus User's Guide. (Order No. DEC-11-LFPUA-A-D.)

[10] DEC PDP-11/70 PDP-11 FORTRAN Language Reference Manual. (Order No. DEC-11-LFLRA-B-D.)

[11] DEC PDP-11/70 IAS Task Builder Reference Manual. (Order No. DEC-11-OITBA-A-D.)

[12] DEC PDP-11/70 ODT Reference Manual. (Order No. DEC-11-010DA-A-D.)

[13] DEC System 10 FORTRAN-10 Programmer's Reference Manual. (Order No. DEC-10-LFORA-D-D.)

[14] DEC System 10 Operating System Commands Manual. (Order No. DEC-10-OSCMA-A-D.)

[15] UNIVAC 1108 Processor and Storage Reference Manual. (UP-4053.)

[16] UNIVAC 1108 Assembler/Programmer's Reference. (UP-4040.)

[17] IBM System 370 OS FORTRAN IV Library Subprograms. (GC28-6596.)

[18] IBM System 370 OS Linkage Editor. (C28-6538.)

[19] Yohe, J. M., "Application of Interval Analysis to Error Control," University of Wisconsin-Madison, Mathematics Research Center, Aug 1976.

[20] ________, "Interval II Package," University of Wisconsin-Madison, Mathematics Research Center, Jul 1976.

[21] Wyatt, W. T., Jr., Lozier, D. W., and Orser, D. J., "A Portable Extended Precision Arithmetic Package and Library with FORTRAN Precompiler," ACM Transactions on Mathematical Software, Vol 2, No. 3, Sep 1976, pp. 209-231.

The following are additional computer listings and runs, for all three computer systems dealt with in the report, they are available from the Automatic Data Processing Center, U. S. Army Engineer Waterways Experiment Station, P. O. Box 631, Vicksburg, Miss. 39180:

AUGMENT Machine Dependent Primitives
Interval II Machine Dependent Primitives
Interval II Machine Dependent Constants
Test Driver for Interval II (supplied by Dr. Yohe)
Test Driver for AUGMENT Primitives (supplied by Dr. Crary)
Factorial Problem
Dr. Yohe's Plot Map Problem
FFT (using normal and interval arithmetic)
Summation of (1/X)**I, I=0,...,128
Roots of a Quadratic Equation
Rounding and Truncation Errors in Addition
Test of SINE, COSINE, and TANGENT Routines
Test of Mathematical Functions
Matrix Inversion
Gaussian Elimination with Partial Pivoting
Gaussian Elimination with No Pivoting
SPLINE
SESOL

APPENDIX A: REMARKS ON SOME ITEMS NOTED IN THIS REPORT

by Dr. Fred D. Crary

The list of remarks that follows is keyed to the main text of the report. Paragraph 1 is the first paragraph beginning on the page. Line 1+ is the first line on the page, 1- is the last line, n+ is the n-th line from the top, and n- is the n-th line from the bottom of the page.

Page 7, para 1: The syntax errors were caused by our faulty keypunching from a correct listing. The implementation of ORDER for the 370 version was suggested by a 370 user. The various "IF's" in the documentation of STRWDS suggested possible implementations of string cells and gave the formulas under these implementations; this description will be revised.

Page 7, para 2: The difficulties with routines MAIN and MOVNUM will be corrected in Version 4L by inserting STOP statements. STOP statements will also be inserted at other points where the called subprogram does not return.

Page 9, para 2: The constant 4250000 was intended to be approximately the largest x such that 10x+9 did not overflow. It appears to have been incorrectly computed (even for full-word integers). Dr. Ward is correct in concluding that this is an overflow test.

Page 9, para 4: The decision table columns other than 12 are for nonoperators and operators which require special handling. Column 12 is for all "normal" single character operators. It happens that "/" is the only standard operator that does not require special handling. If other such operators are defined via the Description Deck, they are allocated to Column 12 also.

Page 10, line 3-: This will be changed to IF (---) J = +1 in Version 4L.

Page 11, line 2+: These warning messages are reasonable responses to the use of the DO index as an argument to a subroutine. In no case is the index altered by the subroutine.

Page 13, line 3-: An attempt will be made to remove the apparent recursion noted.

Page 16, para 1: There is no problem with DOISN because the internal statement numbers begin counting with zero in each program unit and it seems unlikely that 32,000 of them will be needed. There is a possible problem with DOESN; however, this should be caught by the (corrected) overflow test in NUMIN.

Page 19, para 3: Some additional specifics on the tuning desired would be useful.

Page 31, lines 8+ and continuing: It seems as though this problem is easily solved. If the supporting package is written assuming REAL arrays of length 2, simply describe the package to AUGMENT with a "DECLARE REAL (2)" clause rather than "DECLARE COMPLEX".

Page 32, lines 1+ and continuing: ANSI FORTRAN does not recognize tab characters; thus, no provision was made to include them in AUGMENT. EQUIVALENCE and DATA statements are not processed because it is difficult (if not impossible) to determine what effect is intended in all cases. Similarly, the interaction between I/O statements and FORMATs is very difficult to unravel. This is especially true with an essentially one-pass preprocessor.

Page 34, para 1: This observation is inherent in interval arithmetic. If an interval always had an unambiguous sign, AUGMENT would permit the three-branch IF (see pages 54, 62, and 90 of the AUGMENT User Documentation [1]).

Page 35, para 1: This seems to be an overzealous compiler. Is this a warning or a fatal error?

Page 39, para 1: This problem is due to some side effects on the symbol table during the scan of I/O lists for nonstandard variables. The side effects occur only in some cases. A correction is already available and will appear in Version 4L. An interim solution is to use copy mode for all I/O statements (see page 34 of the AUGMENT User Documentation [1]).

Page 39, para 2: The facility requested already exists in the SOURCE option control (see page 30 of the AUGMENT User Documentation [1]). See the output of the test deck supplied with AUGMENT for an example of the output obtained. (A copy of the test deck output is on each microfiche with the AUGMENT revision.)

This comment raises a question about debugging. Do we normally inspect the machine language output of the FORTRAN compiler in order to debug FORTRAN programs? Is this not a similar situation?

In accordance with letter from DAEN-RDC, DAEN-ASI dated 22 July 1977, Subject: Facsimile Catalog Cards for Laboratory Technical Publications, a facsimile catalog card in Library of Congress MARC format is reproduced below.

Ward, Ronnie G
Implementation and evaluation of interval arithmetic software; Report 4: The IBM 370, DEC 10, and DEC PDP-11/70 Systems / by Ronnie G. Ward, Department of Computer Science, University of Texas at Arlington, Arlington, Tex. Vicksburg, Miss. : U. S. Waterways Experiment Station ; Springfield, Va. : available from National Technical Information Service, 1979.
68, 2 p. ; 27 cm. (Technical report - U. S. Army Engineer Waterways Experiment Station ; O-79-1, Report 4)
Prepared for Office, Chief of Engineers, U. S. Army, Washington, D. C., under Contract Nos. DACA39-76-M-0247 and DACA39-76-M-0356.
References: p.67-68.

1. Arithmetic. 2. Computer systems programs. 3. Evaluation. 4. Interval arithmetic. I. Texas. University at Arlington. Dept. of Computer Science. II. United States. Army. Corps of Engineers. III. Series: United States. Waterways Experiment Station, Vicksburg, Miss. Technical report : O-79-1, Report 4.
TA7.W34 no.O-79-1 Report 4